Peer-to-peer

Peer-to-peer is an architectural paradigm used to design distributed systems which was extremely popular in the early 2000s.

In that period of time, hundreds of file sharing applications were developed, which accounted for more than $2/3$ of the entire Internet traffic in 2006.

Peer-to-peer declined over the years due to streaming platforms in particular.

Peer-to-peer is very different from the usual client-server approach. It "takes advantage of the resources at the edges of the network", as Clark Shirky puts it. This change was possible thanks to the incredible increase of resources in end-hosts and the widespread availability of broadband connectivity.

Characteristics of P2P

In P2P, all nodes are potential users and potential providers of a given service. Nodes act as server, clients and routers, depending on needs. Furthermore, each node is independent of the others, so that no central administration is needed.

Systems based on P2P are very dynamic, since its nodes come and go unpredictably; plus, the capabilities of the nodes vary greatly and the scale of the system can even be Internet-wide.

P2P is based on the concept of **overlay network**: a network built on top of another one in order to enhance its capabilities.

Retrieving resources

Retrieving resources is a fundamental issue in P2P systems, due to their inherent geographical distribution. Our problem consists into directing requests towards nodes that can answer them in the most efficient way.

We can distinguish two forms of retrieval operations that can be performed on a data repository:

- Search for something (i.e. "locate all documents about Distributed Systems")
- Lookup a specific item (i.e. "locate a copy of RFC 3268[1]")

Now, we could either retrieve the *actual* data, which could become a burden if the query results are routed through the overlay network and is only meaningful in lookup operations, or a reference to the location from where the data can be retrieved.

Centralised search: Napster

Napster was the first P2P file sharing application. Its key idea was to share the storage and bandwidth of individual users.

Napster used centralised search to look for items in the nodes of its network. It worked like so:

1. **Join**: the client contacts a central server
2. **Publish**: the client submits a list of files to the central server
3. **Search**: the client queries the server for someone owning a specific file
4. **Fetch**: the client gets the file directly from the peer who owns it

Many researchers argued that Napster was not a P2P system, since it depends on server availability. It can be considered an *impure* P2P system.

Regardless, Napster still allowed small computers on the edges of the network to contribute: all peers were active participants to the service, not just consumers.

Having a centralised search has both upsides and downsides:

Pros	Cons
Simple to implement	The server maintains $O(N)$ state
The search scope is constant $O(1)$	The server does all the search processing
	There's a single point of failure
	There's a single point of "control"

Query flooding: Gnutella

Gnutella is another file sharing protocol based on P2P.

It differs from Napster for not having a central authority, so there's no need for users to find a connection point in the network.

Gnutella employs the most basic search algorithm: **flooding**. Each search is forwarded to all neighbours until the requested item is found. Propagation, however, is limited by a `HopsToLive` field in the messages.

To compare Gnutella and Napster, let's take a look at the same steps that we described for Napster here:

1. **Join**: new clients contact a few other nodes, in order to become "neighbours"
2. **Publish**: this step is completely unnecessary in Gnutella
3. **Search**: a client asks all neighbours for a given file; then neighbours do the same until the item is found
4. **Fetch**: the file the client was looking for is downloaded directly from the peer who owns it

Joining the network in Gnutella is done through a well known **anchor** node: the new client who wants to join the network first contacts the anchor, and then sends out a ping message to discover other nodes. Pong messages are sent as a reply to the initial ping message from hosts offering new connections with the new node. Direct connections are then made to newly discovered nodes.

Gnutella has its own set of pros and cons:

Pros	Cons
Fully decentralised	Requests do flood the network
Search cost is distributed	Search scope is $O(N)$
Search can be implemented in many ways (structured database search, text matching, fuzzy finding...)	Search time is $O(2D)$, where D is the average `HopsToLive`
	Nodes leave often, making the network unstable

Hierarchical topology: Kazaa

Kazaa is another P2P file-sharing application from the 2000s.

Kazaa used to distinguish between normal nodes and **supernodes**: the protocol used query flooding among supernodes and normal nodes could contact them in order to perform searches.

Let's look at the usual four steps in Kazaa:

1. **Join**: a new client contacts a supernode[2]

2. **Publish**: the new client sends its files to the supernode
3. **Search**: a client sends a query to a supernode, which in turn floods that same query among other supernodes
4. **Fetch**: the file is downloaded directly from the peer who owns it

Pros	Cons
Tries to consider node heterogeneity	Still no real guarantees on search scope or search time
Kazaa was rumoured to also consider network locality	

Collaborative systems: BitTorrent

A problem which is present in all the above mentioned systems are **free riders**: users who only download files without contributing to the network.

There were some initial attempts at limiting free riders, from user-based ones to credit systems used in EMule and Kazaa, which were all very easy to circumvent.

BitTorrent tries to limit this by allowing many people to download the same file without slowing down everyone else's download. It does this by having downloaders swap portions of a file with one another, instead of all downloading from a single server. This way, each new downloader not only uses up bandwidth, but also contributes bandwidth back to the swarm.

BitTorrent is also the currently most used file-sharing network since 2007.

The steps to join the network and download a file through BitTorrent are:

1. **Join**: a new client contacts a centralised **tracker** server to get a list of peers
2. **Publish**: a client can run a tracker server to publish its own files
3. **Search**: BitTorrent does not support in-band search; instead, users can find torrents elsewhere
4. **Fetch**: once a user finds a torrent, they can download chunks of the file from other peers and reupload chunks of it to the network

Let's now take a look at some BitTorrent terminology:

Term	Description
Torrent	Meta-data file describing the file to be shared. Specifically, it contains the name of the file, its size, the checksum of all of its blocks, the address of the tracker server

Term	Description
	and the address of all the peers hosting that file.
Seed	A peer that has downloaded the complete file and still offers it for upload.
Leech	A peer that has an incomplete download.
Swarm	A group of seeders or leeches.
Tracker	A server that keeps track of seeds and peers in the swarm and gathers statistics.

Content distribution

Let's now understand how content distribution works through BitTorrent.

When a file is uploaded, it is broken down to smaller fragments, usually 256 KB in size. The `.torrent` file holds the SHA1 hash of each fragment, to verify data integrity.

Peers contact the tracker server to have a list of the peers. Then peers download missing file fragments from each other and then upload them to those who don't have it.

It must be noted that the fragments are not downloaded in sequential order and need to be assembled by the receiving machine. When a client needs to choose which segment to request first, it usually adopts a "rarest first" approach, by identifying the fragment held by the fewest of its peers. This tends to keep the number of sources for each segment as high as possible, spreading the load better.

In BitTorrent, clients start uploading whichever fragments they already have **before** the whole download is finished. Once a peer finishes downloading the whole file, it should keep the upload running and become an additional seed in the network.

Everyone can eventually get the complete file, as long as there is one distributed copy of the file in the network, even if there are no seeds.

Choking

Choking is a temporary refusal to upload.

Choking evaluation is performed every 10 seconds and each peer un-chokes a fixed number of peers. The decision on which peers to choke or not is based solely on download rate, which is evaluated on a rolling, 20-second average. This follows the idea that "the more I download from a peer, the higher is the chance I'll also upload to that peer".

A BitTorrent peer also has a single "optimistic un-choke": a peer that is uploaded to regardless of the current download rate from it. The selected peer rotates every 30 seconds. This allows the discovery of currently unused connections that may be better than the ones being used.

Game theory

BitTorrent employs a "tit-for-tat" sharing strategy: "I'll share with you if you share with me".

BitTorrent is also optimistic: it lets freeloaders download occasionally, otherwise no one would ever start a download. This also allows you to discover better peers to download from.

BitTorrent approximates Pareto efficiency: if two peers get poor download rates for the uploads they are providing, they can start uploading to each other and get better download rates than before.

Pros and cons

Finally, let's take a look at BitTorrent's pros and cons:

Pros	Cons
Works reasonably well in practice	Pareto efficiency is a relatively weak condition
Gives peers an incentive to share resources and so avoids free riders	Central tracker servers are needed to bootstrap the swarm

Secure storage: Freenet

Freenet is a P2P application designed to ensure true freedom of communication over the internet. It allows anybody to publish and read information with reasonable anonymity.

Freenet allows the practical, one-to-many publishing of information while providing anonymity for both producers and consumers of information. It does not rely on a centralised control node and it is scalable from tens to thousands of users. It is also robust against node failures or malicious attacks.

This is how a new Freenet user joins the network and fetches a file:

1. **Join**: the new client contacts a few other nodes which they know about and gets a unique node ID
2. **Publish**: the client routes contents toward the node which stores other files whose ID is closest to the file's ID
3. **Search**: the client routes a query for the file ID using a steepest-ascent hill-climbing search with backtracking
4. **Fetch**: when a query reaches a node containing the correct file ID, it returns the file to the sender of the query

Routing protocol

Each node in the network stores some information locally. Nodes also have an approximate knowledge of what their neighbours store.

Requests are forwarded to a node's "best guess" neighbour, unless it has the information locally. If the information is found within the request's hops to live, it's passed back through this chain of nodes to the original requestor. The intermediate nodes store the information in their Least Recently Used (LRU) cache, as it passes through.

Publishing

The insertion of new data in Freenet can be handled similarly to how queries are made.

The inserted data is routed in the same way as a request would:

- Search for the ID of the data to insert
- If the ID is found, the data is not reinserted, since it is already present
- Otherwise, the data is sent along the same path as the request

During searches, data is cached along the way, which guarantees that data migrates towards interested nodes. Each node the resource goes through adds an entry to a routing table associating the key and the data source.

Routing properties

File IDs that are "close" to each other tend to be stored on the same node in Freenet, in order to have similar file IDs be routed towards the same place.

A Freenet network tends to be a "small world": most nodes have relatively few local connections, but a few nodes have many neighbours. Well known nodes tend to see more requests and increase their connections.

Both of these properties make it so that most queries only traverse a small number of hops to find the file.

Caching properties

In Freenet, information tends to migrate towards the areas of demand.

Popular information will be widely cached and files are prioritised according to popularity: unpopular files are even deleted from the disk when it's running out of space. This means that unrequested information may be lost from the network.

Anonymity and security

Since Freenet nodes forward both queries and content back and forth, they cannot tell where a message originated.

File IDs are also created through robust hashing, so that two IDs don't collide and that every file ID is directly related to its content.

If two file IDs could collide easily, denial of service attacks could be performed on the network, censoring information that someone might not want to be spread.

Freenet also offers:

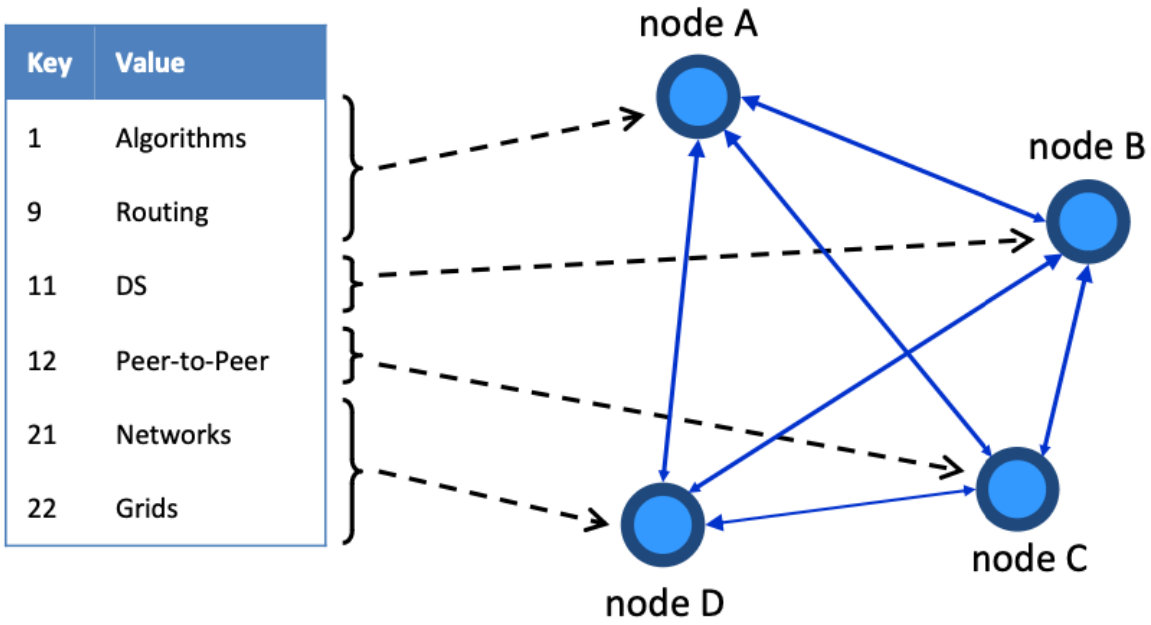
- **Link-level encryption**
  - Prevents the snooping of inter-node messages
  - Messages are quantised to hinder traffic analysis
- **Document encryption**
  - Prevents node operators from knowing what data they are caching
- **Document verification**
  - Allows documents in Freenet to be authenticated
  - It facilitates secure, date-based publishing

Pros and cons

Pros	Cons
Intelligent routing makes queries relatively short	Still no provable guarantees
Small search scope, no flooding involved	Anonymity features make it hard to measure metrics or debug
Anonymity properties may give you plausible deniability	

Structured topology: Distributed Hash Tables (DHTs)

A Distributed Hash Table (or DHT) is similar to the non-distributed data structure, only spread across a whole network:



A DHT still has mainly two primitives:

- `put(id, item)`
- `get(id)`

In this case, items may be any resource, such as a file or a reference to it.

This is how a user joins and searches in a network based on a DHT:

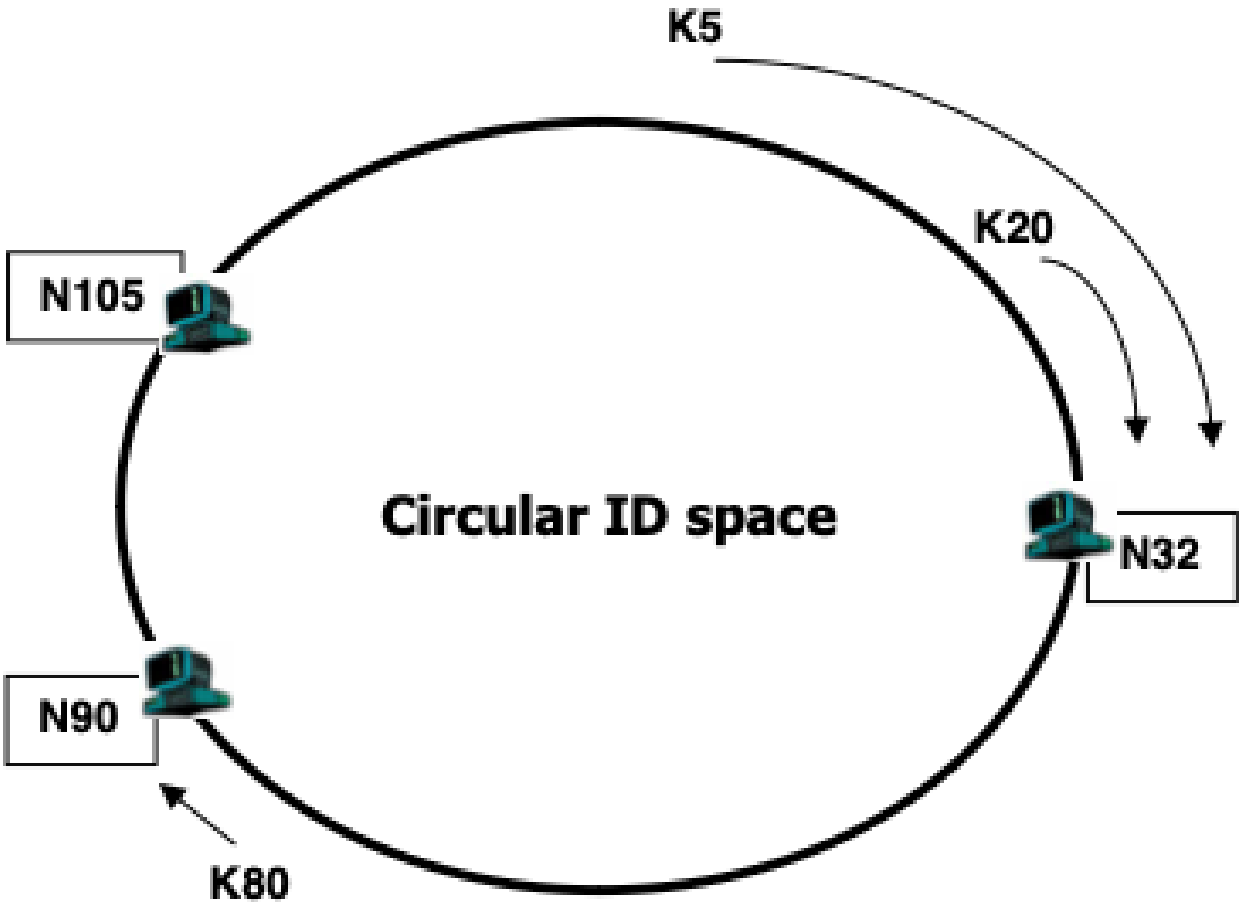
1. **Join**: the new client contacts a bootstrap node and integrates into the distributed data structure, getting a new node ID
2. **Publish**: a client publishes the route for a given file ID toward a close node ID along the data structure

3. **Search**: a client routes a query for a file ID towards a close node ID; the data structure guarantees that the query will meet the publication
4. **Fetch**: if the found publication contains an actual file, it is fetched directly; if it contains a reference, the resource is fetched from the node that owns the file

DHT example: Chord

Chord is a P2P protocol which implements a DHT.

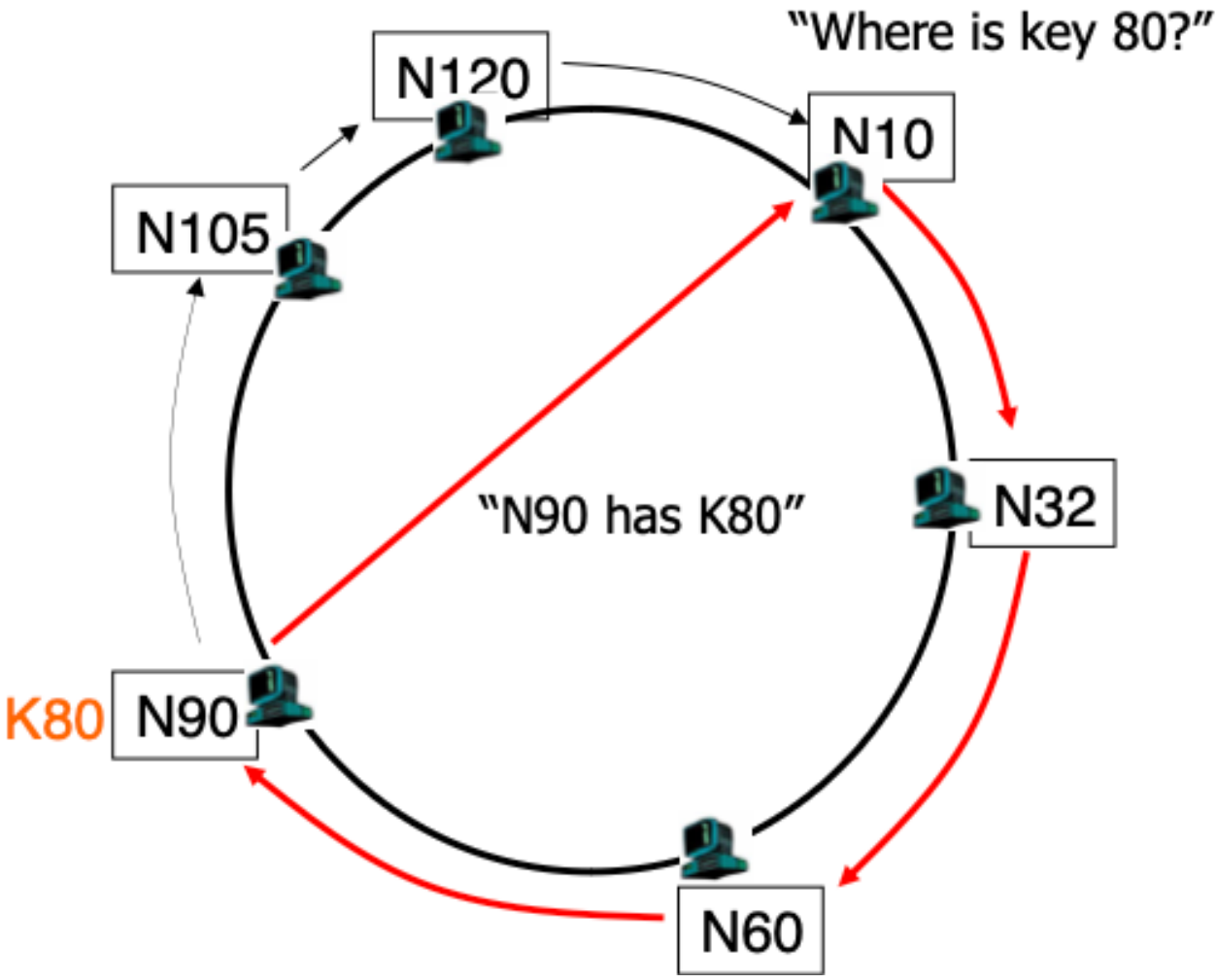
In Chord, nodes and keys are organised in a *logical ring*, in which each node is assigned a unique m-bit identifier (usually the hash of the IP address) and every item is assigned a unique m-bit key (usually the hash of the item).



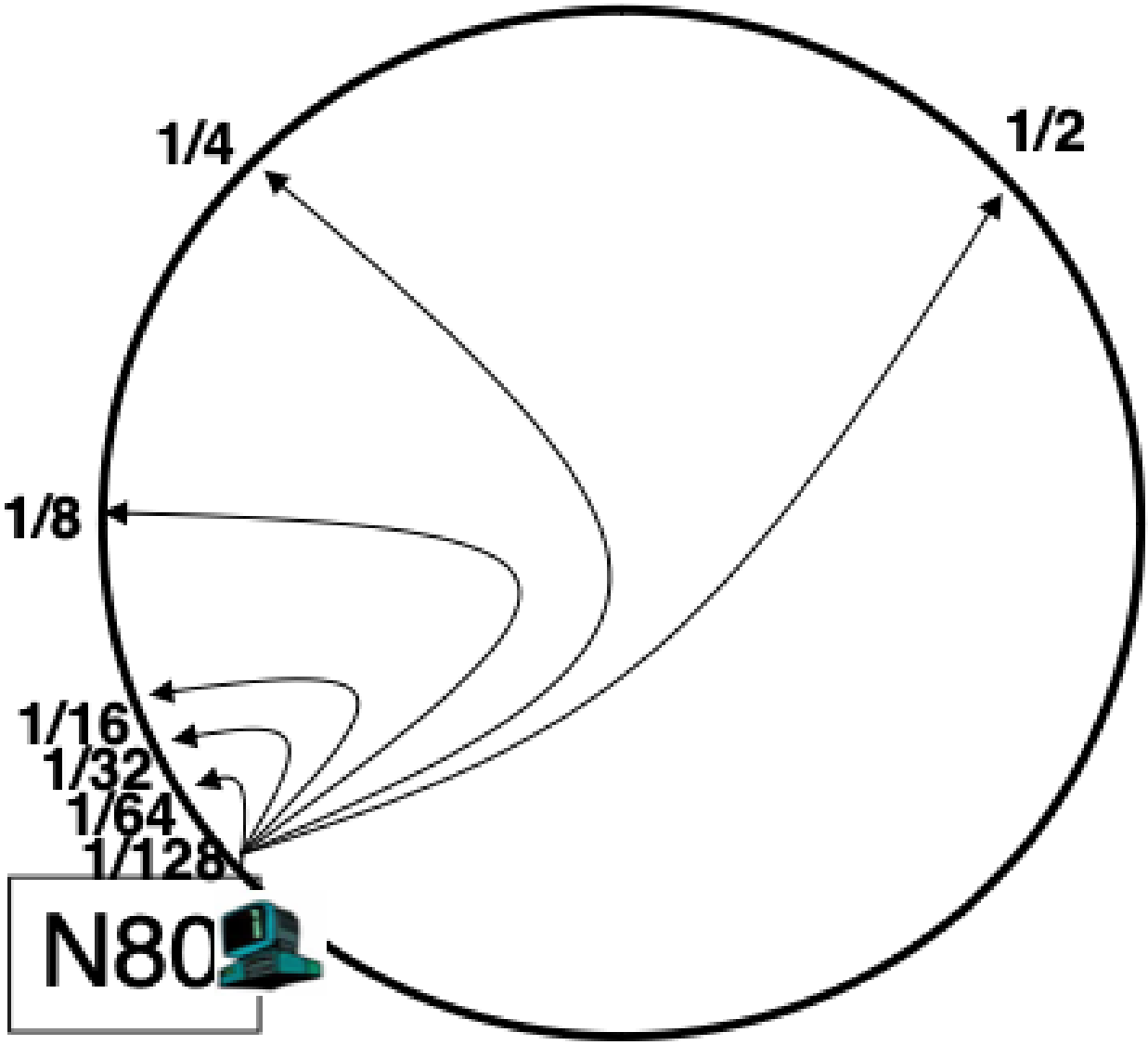
The item with key k is managed by the node with the smallest ID that is greater or equal to k, called **successor**.

Basic lookup

In Chord, each node keeps track of its successor and search is performed linearly:

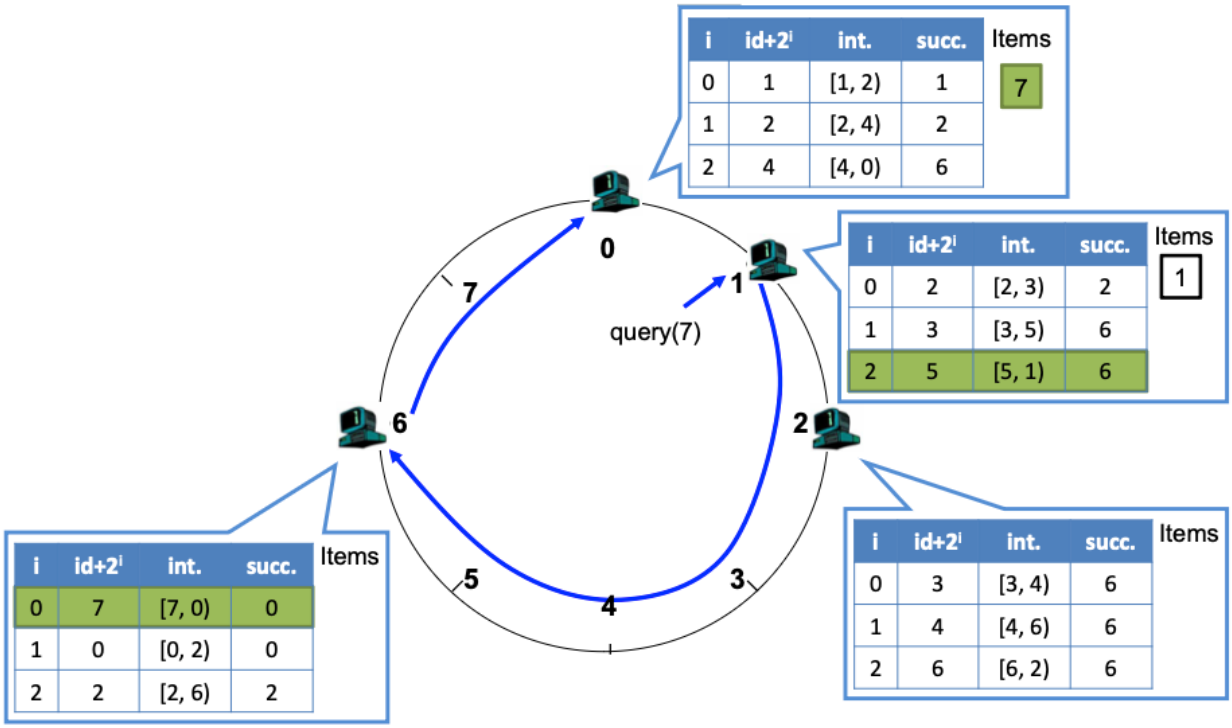


Furthermore, each node maintains a **finger table** with m entries, in which entry i of node n is the first node whose ID is higher or equal to $n + 2^i$. This essentially means that the i-th finger points $1/2^{m-i}$ way around the ring:



Routing

Upon receiving a query for an item with key k, a node checks whether it stores that item locally. If not, it forwards the query to the node in its successor table that is responsible for the interval of keys which includes k.



Joining

In Chord, each node also keeps track of its predecessor, to allow for counter-clockwise routing, useful to manage join operations. When a new node n joins, the following actions must be performed:

1. Initialise the predecessor and fingers of node n
2. Update the fingers and predecessors of existing nodes to reflect the addition of n

To initialise its predecessor and fingers, we assume n knows another node n' already into the system, in which case it uses n' to initialise its fingers.

To update fingers of other nodes, we observe that node n will become the i-th finger of node p if and only if p precedes n by at least 2^{i-1} and the i-th finger of p succeeds n. The first node p that meets these two conditions is the immediate predecessor of node $n - 2i - 1$, so the system searches for this node and updates it.

We need to perform this operation for each finger i, which means a complexity of $\log(N)$ fingers times a lookup that costs $\log(N)$, resulting in a total cost of $\log^2(N)$.

Stabilisation

The correctness of Chord relies on the correctness of successor pointers. This may not be preserved in the case of multiple nodes joining and leaving simultaneously,

so Chord periodically runs a stabilisation procedure to ensure this invariant. The stabilisation procedure consists in the periodic lookup of the successor node $n + 2^{i-1}$ for every node n in the system and for every finger i in the node.

Failure and replication

To increase robustness, each Chord node maintains a list of successors of size R so that, if the node's immediate successor does not respond, the node contacts the next in the list.

Pros and cons

Pros	Cons
Guaranteed lookup	No one uses this protocol
$O(\log N)$ per node state and search scope	It's more fragile than unstructured networks
	Supporting non-exact match search is hard
	It does not consider physical topology

Other protocols

Chord is just an example: other structures exist, with different tradeoffs between search scope, information to store, cost to maintain the structure...

Some more examples are:

- **Kademila**: tree-based structure
- **CAN**: d-dimensional space, in which the number of dimensions can be changed to better adapt the system to the application at hand

DHTs also power the Interplanetary File System, which uses them to create a content-based infrastructure.

---

1. This is the standard for AES ↩
2. Clients could also become supernodes at some point ↩